



Parallel programming models, tools and performance analysis

Michael Gerndt

published in

Modern Methods and Algorithms of Quantum Chemistry,
J. Grotendorst (Ed.), John von Neumann Institute for Computing,
Jülich, NIC Series, Vol. 1, ISBN 3-00-005618-1, pp. 9-27, 2000.

© 2000 by John von Neumann Institute for Computing

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

<http://www.fz-juelich.de/nic-series/>

PARALLEL PROGRAMMING MODELS, TOOLS AND PERFORMANCE ANALYSIS

MICHAEL GERNDT

*John von Neumann Institute for Computing
Central Institute for Applied Mathematics
52425 Jülich
Germany
E-mail: m.gerndt@fz-juelich.de*

The major parallel programming models for scalable parallel architectures are the message passing model and the shared memory model. This article outlines the main concepts of those models as well as the industry standard programming interfaces MPI and OpenMP. To exploit the potential performance of parallel computers, programs need to be carefully designed and tuned. We will discuss design decisions for good performance as well as programming tools that help the programmer in program tuning.

1 Introduction

Although the performance of sequential computers increases incredibly fast, it is insufficient for a large number of challenging applications. Applications requiring much more performance are numerical simulations in industry and research as well as commercial applications such as query processing, data mining, and multi-media applications. Architectures offering high performance do not only exploit parallelism on a very fine grain within a single processor but apply a medium to large number of processors concurrently to a single application. High end parallel computers deliver up to 3 Teraflop/s (10^{12} floating point operations per second) and are developed and exploited within the ASCI (Accelerated Strategic Computing Initiative) program of the Department of Energy in the USA.

This article concentrates on programming numerical applications on distributed memory computers introduced in Section 1.1. Parallelization of those applications centers around selecting a decomposition of the data domain onto the processors such that the workload is well balanced and the communication is reduced (Section 1.2)⁷.

The parallel implementation is then based on either the message passing or the shared memory model (Section 2). The standard programming interface for the message passing model is MPI (Message Passing Interface)^{13,10}, offering a complete set of communication routines (Section 2.1). OpenMP^{5,12} is the standard for directive-based shared memory programming and will be introduced in Section 2.2.

Since parallel programs exploit multiple threads of control, debugging is even more complicated than for sequential programs. Section 3 outlines the main concepts of parallel debuggers and presents TotalView, the most widely available debugger for parallel programs.

Although the domain decomposition is key to good performance on parallel architectures, program efficiency depends also heavily on the implementation of the

communication and synchronization required by the parallel algorithm and the implementation techniques chosen for sequential kernels. Optimizing those aspects is very system dependent and thus, an interactive tuning process consisting of measuring performance data and applying optimizations follows the initial coding of the application. The tuning process is supported by programming model specific performance analysis tools. Section 4 presents basic performance analysis techniques and introduces the three performance analysis tools for MPI programs available on CRAY T3E.

1.1 Parallel Architectures

Parallel computers that scale beyond a small number of processors circumvent the main memory bottleneck by distributing the memory among the processors. Current architectures⁴ are composed of single-processor nodes with local memory or of multiprocessor nodes where the node's main memory is shared among the node's processors. In the following it is assumed that nodes do have only a single CPU and the terms node and processor will be used interchangeably.

The most important characteristic of this *distributed memory architecture* is that access to the local memory is faster than to remote memory. It is the challenge for the programmer to assign data to the processors such that most of the data accessed during the computation are already in the node's local memory.

Three major classes of distributed memory computers can be distinguished:

No Remote Memory Access (NORMA) computers do not have any hardware support to access another node's local memory. Processors obtain data from remote memory only by exchanging messages between processes on the requesting and the supplying node.

Remote Memory Access (RMA) computers allow to access remote memory via specialized operations implemented by hardware. The accessed memory location is not determined via an address in a shared linear address space but via a tuple consisting of the processor number and the local address in the target processor's address space.

Cache-Coherent Non Uniform Memory Access (ccNUMA) computers do have a shared physical address space. All memory locations can be accessed via usual load and store operations. Access to a remote location results in a copy of the appropriate cache line in the processor's cache. Coherence algorithms ensure that multiple copies of a cache line are kept coherent, i.e. the copies do have the same value.

While most of the early parallel computers were NORMA systems, today's systems are either RMA or ccNUMA computers. This is because remote memory access is a light-weight communication protocol that is more efficient than standard message passing since data copying and process synchronization are eliminated. In addition, ccNUMA systems offer the abstraction of a shared linear address space resembling physical shared memory systems. This abstraction simplifies the task of program development but does not necessarily facilitate program tuning.

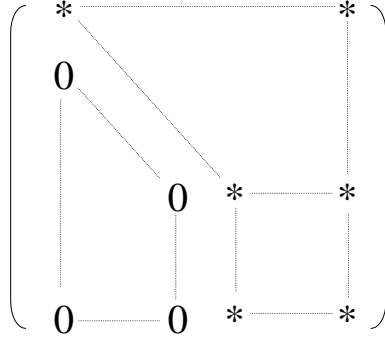


Figure 1. Structure of the matrix during Gaussian elimination.

Typical examples of the three classes are clusters of workstations (NORMA), CRAY T3E (RMA), and SGI Origin 2000 (ccNUMA).

1.2 Data Parallel Programming

Applications that scale to a large number of processors usually perform computations on large data domains. For example, crash simulations are based on partial differential equations that are solved on a large finite element grid and molecular dynamic applications simulate the behavior of a large number of atoms. Other parallel applications apply linear algebra operations to large vectors and matrices. The elemental operations on each object in the data domain can be executed in parallel by the available processors.

The scheduling of operations to processors is determined according to a selected *domain decomposition*⁸. Processors execute those operations that determine new values for local elements (owner-computes rule). While processors execute an operation, they might need values from other processors. The domain decomposition has thus to be chosen so that the distribution of operations is balanced and the communication is minimized. The third goal is to optimize single node computation, i.e. to be able to exploit the processor's pipelines and the processor's caches efficiently.

A good example for the design decisions taken when selecting a domain decomposition is Gaussian elimination². The main structure of the matrix during the iterations of the algorithm is outlined in Figure 1.

The goal of this algorithm is to eliminate all entries in the matrix below the main diagonal. It starts at the top diagonal element and subtracts multiples of the first row from the second and subsequent rows to end up with zeros in the first column. This operation is repeated for all the rows. In later stages of the algorithm the actual computations have to be done on rectangular sections of decreasing size.

If the main diagonal element of the current row is zero, a pivot operation has to be performed. The subsequent row with the maximum value in this column is selected and exchanged with the current row.

A possible distribution of the matrix is to decompose its columns into blocks, one

block for each processor. The elimination of the entries in the lower triangle can then be performed in parallel where each processor computes new values for its columns only. The main disadvantage of this distribution is that in later computations of the algorithms only a subgroup of the processes is actually doing any useful work since the computed rectangle is getting smaller.

To improve load balancing, a cyclic column distribution can be applied. The computations in each step of the algorithm executed by the processors differ only in one column.

In addition to load balancing also communication needs to be minimized. Communication occurs in this algorithm for broadcasting the current column to all the processors since it is needed to compute the multiplication factor for the row. If the domain decomposition is a row distribution, which eliminates the need to communicate the current column, only the main diagonal element needs to be broadcast to the other processors.

If we consider also the pivot operation, communication is necessary to select the best row when a rowwise distribution is applied since the computation of the global maximum in that column requires a comparison of all values.

Selecting the best domain decomposition is further complicated due to optimizing single node performance. In this example, it is advantageous to apply BLAS3 operations for the local computations. Those operations make use of blocks of rows to improve cache utilization. Blocks of rows can only be obtained if a block-cyclic distribution is applied, i.e. columns are not distributed individually but blocks of columns are cyclically distributed.

This discussion makes clear, that choosing a domain decomposition is a very complicated step in program development. It requires deep knowledge of the algorithm's data access patterns as well as the ability to predict the resulting communication.

2 Programming Models

The two main programming models, *message passing* and *shared memory*, offer different features for implementing applications parallelized by domain decomposition.

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. The domain decomposition is implemented by developing a code describing the local computations and local data structures of a single process. Thus, global arrays have to be split up and only the local part be allocated in a process. This handling of global data structures is called *data distribution*. Computations on the global arrays also have to be transformed, e.g. by adapting the loop bounds, to ensure that only local array elements are computed. Access to remote elements have to be implemented via explicit communication, temporary variables have to be allocated, messages be setup and transmitted to the target process.

The shared memory model is based on a set of threads that are created when parallel operations are executed. This type of computation is also called *fork-join*

parallelism. Threads share a global address space and thus access array elements via a global index.

The main parallel operations are *parallel loops* and *parallel sections*. Parallel loops are executed by a set of threads also called *team*. The iterations are distributed onto the threads according to a predefined strategy. This scheduling strategy implements the chosen domain decomposition. Parallel sections are also executed by a team of threads but the tasks assigned to the threads implement different operations. This feature can for example be applied if domain decomposition itself does not generate enough parallelism and whole operations can be executed in parallel since they access different data structures.

In the shared memory model, the distribution of data structures onto the node memories is not enforced by decomposing global arrays into local arrays, but the global address space is distributed onto the memories on system level. For example, the pages of the virtual address space can be distributed cyclically or can be assigned on a first touch basis. The chosen domain decomposition thus has to take into account the granularity of the distribution, i.e. the size of pages, as well as the system-dependent allocation strategy.

While the domain decomposition has to be hardcoded into the message passing program, it can easily be changed in a shared memory program by selecting a different scheduling strategy for parallel loops.

Another advantage of the shared memory model is that automatic and incremental parallelization is supported. While automatic parallelization leads to a first working parallel program, its efficiency typically needs to be improved. The reason for this is that parallelization techniques work on a loop-by-loop basis and do not globally optimize the parallel code via a domain decomposition. In addition, dependence analysis, the prerequisite for automatic parallelization, is limited to statically known access patterns.

In the shared memory model, a first parallel version is relatively easy to implement and can be incrementally tuned. In the message passing model instead, the program can be tested only after finishing the full implementation. Subsequent tuning by adapting the domain decomposition is usually time consuming.

2.1 MPI

The Message Passing Interface (MPI) ^{13,10} was developed between 1993 and 1997. It includes routines for point-to-point communication, collective communication, one-sided communication, and parallel IO. While the basic communication primitives are already defined since May 1994 and implemented on almost all parallel computers, remote memory access and parallel IO routines are part of MPI 2.0 and are only available on few machines.

2.1.1 MPI basic routines

MPI consists of more than 120 functions. But realistic programs can already be developed based on no more than six functions:

MPI_Init initializes the library. It has to be called at the beginning of a parallel operation before any other MPI routines are executed.

MPI_Finalize frees any resources used by the library and has to be called at the end of the program.

MPI_Comm_size determines the number of processors executing the parallel program.

MPI_Comm_rank returns the unique process identifier.

MPI_Send transfers a message to a target process. This operation is a blocking send operation, i.e. it terminates when the message buffer can be reused either because the message was copied to a system buffer by the library or because the message was delivered to the target process.

MPI_Recv receives a message. This routine terminates if a message was copied into the receive buffer.

2.1.2 MPI communicator

All communication routines depend on the concept of a *communicator*. A communicator consists of a process group and a communication context. The processes in the process group are numbered from zero to process count - 1. The process number returned by **MPI_Comm_rank** is the identification in the process group of the communicator which is passed as a parameter to this routine.

The communication context of the communicator is important in identifying messages. Each message has an integer number called a *tag* which has to match a given selector in the corresponding receive operation. The selector depends on the communicator and thus on the communication context. It selects only messages with a fitting tag and having been sent relative to the same communicator. This feature is very useful in building parallel libraries since messages sent inside the library will not interfere with messages outside if a special communicator is used in the library. The default communicator that includes all processes of the application is **MPI_COMM_WORLD**.

2.1.3 MPI collective operation

Another important class of operations are *collective operations*. Collective operations are executed by a process group identified via a communicator. All the processes in the group have to perform the same operation. Typical examples for such operations are:

MPI_Reduce performs a global operation on the data of each process in the process group. For example, the sum of all values of a distributed array can be computed by first summing up all local values in each process and then summing up the local sums to get a global sum. The latter step can be performed by the reduction operation with the parameter **MPI_SUM**. The result is delivered to a single target processor.

MPI_Comm_split is an administration routine for communicators. It allows to create multiple new communicators based on a given coloring scheme. All processes of the original communicator have to take part in that operation.

MPI_Barrier synchronizes all processes. None of the processes can proceed beyond the barrier until all the processes started execution of that routine.

2.1.4 MPI IO

Data parallel applications make use of the IO subsystem to read and write big data sets. These data sets result from replicated or distributed arrays. The reasons for IO are to read input data, to pass information to other programs, e.g. for visualization, or to store the state of the computation to be able to restart the computation in case of a system failure or if the computation has to be split into multiple runs due to its resource requirements.

IO can be implemented in three ways:

1. Sequential IO

A single node is responsible to perform the IO. It gathers information from the other nodes and writes it to disc or reads information from disc and scatters it to the appropriate nodes. While the IO is sequential and thus need not be parallelized, the full performance of the IO subsystem might not be utilized. Modern systems provide high performance IO subsystems that are fast enough to support multiple IO requests from different nodes in parallel.

2. Private IO

Each node accesses its own files. The big advantage of this implementation is that no synchronization among the nodes is required and very high performance can be obtained. The major disadvantage is that the user has to handle a large number of files. For input the original data set has to be splitted according to the distribution of the data structure and for output the process-specific files have to be merged into a global file for postprocessing.

3. Parallel IO

In this implementation all the processes access the same file. They read and write only those parts of the file with relevant data. The main advantages are that no individual files need to be handled and that reasonable performance can be reached. The disadvantage is that it is difficult to reach the same performance as with private IO. The parallel IO interface of MPI provides flexible and high-level means to implement applications with parallel IO.

Files accessed via MPI IO routines have to be opened and closed by collective operations. The open routine allows to specify hints to optimize the performance such as whether the application might profit from combining small IO requests from different nodes, what size is recommended for the combined request, and how many nodes should be engaged in merging the requests.

The central concept in accessing the files is the *view*. A view is defined for each process and specifies a sequence of data elements to be ignored and data elements

to be read or written by the process. When reading or writing a distributed array the local information can be described easily as such a repeating pattern. The IO operations read and write a number of data elements on the basis of the defined view, i.e. they access the local information only. Since the views are defined via runtime routines prior to the access, the information can be exploited in the library to optimize IO.

MPI IO provides blocking as well as nonblocking operations. In contrast to blocking operations, the nonblocking ones only start IO and terminate immediately. If the program depends on the successful completion of the IO it has to check it via a test function. Besides the collective IO routines which allow to combine individual requests, also noncollective routines are available to access shared files.

2.1.5 MPI remote memory access

Remote memory access (RMA) operations allow to access the address space of other processes without participation of the other process. The implementation of this concept can either be in hardware, such as in the CRAY T3E, or in software via additional threads waiting for requests. The advantages of these operations are that the protocol overhead is much lower than for normal send and receive operations and that no polling or global communication is required for setting up communication such as in unstructured grid applications and multiparticle applications.

In contrast to explicit message passing where synchronization happens implicitly, accesses via RMA operations need to be protected by explicit synchronization operations.

RMA communication in MPI is based on the *window concept*. Each process has to execute a collective routine that defines a window, i.e. the part of its address space that can be accessed by other processes.

The actual access is performed via a *put* and *get operation*. The address is defined by the target process number and the displacement relative to the starting address of the window for that process.

MPI provides also special synchronization operations relative to a window. The `MPI_Win_fence` operation synchronizes all processes that make some address ranges accessible to other processes. It is a collective operation that ensures, that all RMA operations started before the fence operation terminate before the target process executes the fence operation and that all RMA operations of a process executed after the fence operation are executed after the target process executed the fence operation.

2.2 OpenMP

OpenMP^{5,12} is a directive-based programming interface for the shared memory programming model. It is the result of an effort to standardize the different programming interfaces on the target systems. OpenMP is a set of directives and runtime routines for Fortran 77 (1997) and a corresponding set of pragmas for C and C++ (1998). An extension of OpenMP for Fortran 95 is under investigation.

Directives are comments that are interpreted by the compiler. Directives do have the advantage that the code is still a sequential code that can be executed on

sequential machines and thus no two versions, a sequential and a parallel version, need to be maintained.

Directives start and terminate parallel regions. When the master thread hits a parallel region a team of threads is created or activated. The threads execute the code in parallel and are synchronized at the beginning and the end of the computation. After the final synchronization the master thread continues execution after the parallel region. The main directives are:

PARALLEL DO specifies a loop that can be executed in parallel. The DO loop's iterations can be distributed in various ways including **STATIC(CHUNK)**, **DYNAMIC(CHUNK)**, and **GUIDED(CHUNK)** onto the set of threads (as defined in the OpenMP standard). **STATIC(CHUNK)** distribution means that the set of iterations are consecutively distributed onto the threads in blocks of **CHUNK** size (resulting in block and cyclic distributions). **DYNAMIC(CHUNK)** distribution implies that iterations are distributed in blocks of **CHUNK** size to threads on a first-come-first-served basis. **GUIDED (CHUNK)** means that blocks of exponentially decreasing size are assigned on a first-come-first-served basis. The size of the smallest block is determined by **CHUNK** size.

PARALLEL SECTION starts a set of sections that are executed in parallel by a team of threads.

PARALLEL REGION introduces a code region that is executed redundantly by the threads. It has to be used very carefully since assignments to global variables will lead to conflicts among the threads and possibly to nondeterministic behavior.

PDO is a work sharing construct and may be used within a parallel region. All the threads executing the parallel region have to cooperate in the execution of the parallel loop. There is no implicit synchronization at the beginning of the loop but a synchronization at the end. After the final synchronization all threads continue after the loop in the replicated execution of the program code.

The main advantage of this approach is that the overhead for starting up the threads is eliminated. The team of threads exists during the execution of the parallel region and need not be built before each parallel loop.

PSECTION is also a work sharing construct that enforces that the current team of threads executing the surrounding parallel region cooperates in the execution of the parallel section.

Program data can either be shared or private. While threads do have an own copy of private data, only one copy exists of shared data. This copy can be accessed by all threads. To ensure program correctness, OpenMP provides special synchronization constructs. The main constructs are *barrier synchronization* enforcing that all threads have reached this synchronization operation before execution continues and *critical sections*. Critical sections ensure that only a single thread can enter the section and thus, data accesses in such a section are protected from race conditions. A common situation for a critical section is the accumulation of values. Since an

accumulation consists of a read and a write operation unexpected results can occur if both operations are not surrounded by a critical section.

3 Parallel Debugging

Debugging parallel programs is more difficult than debugging sequential programs not only since multiple processes or threads need to be taken into account but also because program behaviour might not be deterministic and might not be reproducible. These problems are not attacked by current state-of-the-art commercial parallel debuggers. Only the first reason is elivated by current debuggers. They provide menus, diplays, and commands that allow to inspect individual processes and execute commands on individual or all processes.

The widely used debugger is TotalView from Etnus Inc ¹⁴. TotalView provides breakpoint definition, single stepping, and variable inspection via an interactive interface. The programmer can execute those operations for individual processes and groups of processes. TotalView also provides some means to summarize information such that equal information from multiple processes is combined into a single information and not repeated redundantly.

4 Performance Analysis

Performance analysis is an iterative subtask during process development. The goal is to identify program regions that do not perform well. Performance analysis is structured into four phases:

1. Measurement

Performance analysis is executed based on information on runtime events gathered during program execution. The basic events are, for example, cache misses, termination of a floating point operation, start and stop of a subroutine or message passing operation. The information on individual events can be summarized during program execution or individual trace records can be collected for each event.

Summary information has the advantage to be of moderate size while trace information tends to be very large. The disadvantage is that it is not that fine grained, the behavior of individual instances of subroutines can for example not be investigated since all the information has been summed up.

2. Analysis

During analysis the collected runtime data are inspected to detect *performance problems*. Performance problems are based on *performance properties*, such as the existence of message passing in a program region, which do have a condition for identifying it and a severity function that specifies its importance for program performance.

Current tools support the user in checking the conditions and the severity by visualizing program behavior. Future tools might be able to automatically detect performance properties based on a specification of possible properties.

During analysis the programmer applies a threshold. Only performance properties whose severity exceeds this threshold are considered to be performance problems.

3. Ranking

During program analysis the severest performance problems need to be identified. This means that the problems need to be ranked according to the severity. The most severe problem is called the *program bottleneck*. This is the problem the programmer tries to resolve by applying appropriate program transformations.

4. Refinement

The performance problems detected in the previous phases might not be precise enough to allow the user to start optimization. At the beginning of performance analysis, summary data can be used to identify critical regions. Those summary data might not be sufficient to identify why, for example, a region has high message passing overhead. The reason, e.g. very big messages or load imbalance, might be identified only with more detailed information. Therefore the performance problem should be refined into hypotheses about the real reason and additional information be collected in the next performance analysis cycle.

Current techniques for performance data collection are *profiling* and *tracing*. Profiling collects summary data only. This can be done via *sampling*, the program is regularly interrupted, e.g. every 10 ms, and the information is added up for the source code location which was executed in this moment. For example, the UNIX profiling tool *prof* applies this technique to determine the fraction of the execution time spent in individual subroutines.

A more precise profiling technique is based on *instrumentation*, i.e. special calls to a *monitoring library* are inserted into the program. This can either be done in the source code by the compiler or specialized tools, or can be done in the object code. While the first approach allows to instrument more types of regions, for example, loops and vector statements, the latter allows to measure data for programs where no source code is available. The monitoring library collects the information and adds it to special counters for the specific region.

Tracing is a technique that collects information for each event. This results, for example, in very detailed information for each instance of a subroutine and for each message sent to another process. The information is stored in specialized trace records for each event type. For example, for each start of a send operation, the time stamp, the message size and the target process can be recorded, while for the end of the operation, the timestamp and bandwidth are stored.

The trace records are stored in the memory of each process and are output either because the buffer is filled up or because the program terminated. The individual trace files of the processes are merged together into one trace file ordered according to the time stamps of the events.

The following sections describe performance analysis tools available on CRAY T3E.

4.1 The CRAY T3E Performance Analysis Environment

The programming environment of the CRAY T3E supports performance analysis via interactive tools. Cray itself provides two tools, Apprentice and PAT, which are both based on summary information. In addition, Apprentice accesses source code information to map the performance information to the statements of the original source code. Besides these two tools, programmers can use VAMPIR, a trace analysis tool developed at our institute. Within a collaboration with Cray, instrumentation and trace generation for VAMPIR is being integrated into the next version of PAT.

4.2 Compiler Information File

The F90 and C compilers on CRAY T3E generate for each source file on request a *compiler information file (CIF)*. This file includes information about the compilation process (applied compiler options, target machine characteristics, and compiler messages) and source information of the compilation units (procedure information, symbol information, information about loops and statement types, cross-reference information for each symbol in the source file).

Apprentice requires this information to link the performance information back to the source code. CIFs are initially in ASCII format but can be converted to binary format. The information can be easily accessed in both formats via a library interface.

4.3 Apprentice

Apprentice is a post-execution performance analysis tool for message passing programs³. Originally it was designed to support the CRAFT programming model on the CRAY T3E predecessor system, the CRAY T3D. Apprentice analyzes summary information collected at runtime via an instrumentation of the source program.

The instrumentation is performed by the compiler and is triggered via an appropriate compiler switch. To reduce the overhead of the instrumentation, the programmer can selectively compile the source files with and without instrumentation. The instrumentation is done in a late phase of the compilation after all optimizations already occurred. This prevents that instrumentation affects the way code is compiled. During runtime, summary information is collected at each processor for each basic block. This information comprises:

- execution time
- number of floating point, integer, and load/store operations
- instrumentation overhead

For each subroutine call the execution time as well as the pass count is determined. At the end of a program run, the information of all processors is summed up and written to the *runtime information file (RIF)*. In addition to the summed up execution times and pass counts of subroutines calls, their mean value and standard deviation, as well as the minimum and maximum values are stored.

The size of the resulting RIF is typically less than one megabyte. But the overhead due to the instrumentation can easily be a factor of two which results from instrumenting each basic block. This severe drawback of the instrumentation is partly compensated in Apprentice by correcting the timings based on the measured overhead.

When the user starts Apprentice to analyze the collected information, the tool first reads the RIF as well as the CIFs of the individual source files. The performance data measured for the optimized code are related back to the original source code. Apprentice distinguishes between:

- parallel work: user-level subroutines
- I/O: system subroutines for performing I/O
- communication overhead: MPI and PVM routines, SHMEM routines
- uninstrumented code

The available barcharts allow the user to identify critical code regions that take most of the execution time or with a lot of I/O and communication overhead. Since all the values have been summed up, no specific behavior of the processors can be identified. Load balance problems can be detected by inspecting the execution times of calls to synchronization subroutines, such as global sums or barriers. Based on the available information, the processors with the least and the highest execution time can be identified.

While Apprentice does not evaluate the hardware performance counters of the DEC Alpha, it estimates the loss due to cache misses and suboptimal use of the functional units. Based on the number of instructions and a very simple cost model (fixed cycles for each type of instruction) it determines the loss as the difference between the estimated optimal and the measured execution time.

4.4 VAMPIR

VAMPIR (Visualization and Analysis of MPI Resources) is an event trace analysis tool¹¹ which was developed by the Central Institute for Applied Mathematics of the Research Centre Jülich and now is commercially distributed by a German company named PALLAS. Its main application area is the analysis of parallel programs based on the message passing paradigm but it also has been successfully used for other areas (e.g., for SVM-Fortran traces to analyze shared virtual memory page transfer behavior⁹ or to analyze CRAY T3E usage based on accounting data). VAMPIR has three components:

- The VAMPIR tool itself is a graphical event trace browser implemented for the X11 Window system using the Motif toolkit. It is available for any major UNIX platform.
- The VAMPIR runtime library provides an API for collecting, buffering, and generating event traces as well as a set of wrapper routines for the most commonly used MPI and PVM communication routines which record message traffic in the event trace.

- In order to observe functions or subroutines in the user program, their entry and exit has to be instrumented by inserting calls to the VAMPIR runtime library. Observing message passing functions is handled by linking the program with the VAMPIR wrapper function library.

VAMPIR comes with a source instrumenter for ANSI Fortran 77. Programs written in other programming languages (e.g., C or C++) have to be instrumented manually. To improve this situation, our institute in collaboration with CRAY Research is currently implementing an object code instrumenter for CRAY T3E. This is described in the next section.

During the execution of the instrumented user program, the VAMPIR runtime library records entry and exits to instrumented user and message passing functions and the sending and receiving of messages. For each message, its tag, communicator, and length is recorded. Through the use of a configuration file, it is possible to switch the runtime observation of specific functions on and off. This way, the program doesn't have to be re-instrumented and re-compiled for every change in the instrumentation.

Large parallel programs consist of several dozens or even hundreds of functions. To ease the analysis of such complex programs, VAMPIR arranges the functions into groups, e.g., user functions, MPI routines, I/O routines, and so on. The user can control/change the assignment of functions to groups and can also define new groups.

VAMPIR provides a wide variety of graphical displays to analyze the recorded event traces:

- The dynamic behavior of the program can be analyzed by timeline diagrams for either the whole program or a selected set of nodes. By default, the displays show the whole event trace, but the user can zoom-in to any arbitrary region of the trace. Also, the user can change the display style of the lines representing messages based on their tag/communicator or the length. This way, message traffic of different modules or libraries can easily be visually separated.
- The parallelism display shows the number of nodes in each function group over time. This allows to easily locate specific parts of the program, e.g., parts with heavy message traffic or I/O.
- VAMPIR also provides a large number of statistical displays. It calculates how often each function or group of functions got called and the time spent in there. Message statistics show the number of messages sent, and the minimum, maximum, sum, and average length or transfer rate between any two nodes. The statistics can be displayed as barcharts, histograms, or textual tables.

A very useful feature of VAMPIR is that the statistic displays can be linked to the timeline diagrams. By this, statistics can be calculated for any arbitrary, user selectable part of the program execution.

- If the instrumenter/runtime library provides the necessary information in the event trace header, the information provided by VAMPIR can be related back

to source code. VAMPIR provides a source code and a call graph display to show selected functions or the location of the send and the receive of a selected message.

In summary, VAMPIR is a very powerful and highly configurable event trace browser. It displays trace files in a variety of graphical views, and provides flexible filter and statistical operations that condense the displayed information to a manageable amount. Rapid zooming and instantaneous redraw allow to identify and focus on the time interval of interest.

4.5 PAT

PAT (Performance Analysis Tool) is the second performance tool available from CRAY Research for CRAY T3E. The two main differences to Apprentice are that no source code instrumentation or special compiler support is necessary. The user only needs to re-link his/her application against the PAT runtime library (because CRAY Unicos doesn't support dynamic linking). Second, PAT aims at keeping the additional overhead to measure/observe program behavior as low as possible. PAT is actually three performance tools in one:

1. PAT allows the user to get an rough overview about the performance of the parallel program through a method called sampling, i.e., interrupting the program at regular intervals and evaluating the program counter. PAT can calculate then the percentage of time spent in each function. The sampling rate can be changed by the user to adapt it to the execution time of the program and to keep overhead low. Because the sampling method provides only a statistical estimate of the actual time spent in a function, the tool also provides a measure of confidence in the sampling estimate.

In addition PAT determines the total, user, and system time of the execution run and the number of cache misses and the number of either floating point, integer, store, or load operations. These are measured through the DEC Alpha hardware counters. The user can select the hardware counter by setting an environment variable. All this statistical information is stored after the execution in a so-called *Performance Information File* (PIF).

2. If a more detailed analysis is necessary, PAT can be used to instrument and analyze a specific function or set of functions in a second phase. PAT can instrument object code (however only on the function level). This is a big advantage especially for large complex programs because they do not have to be re-compiled for instrumentation. In addition, it is possible to analyze functions contained in system or 3rd-party libraries. A third advantage is that programs written in more than one language can be handled. The big disadvantage is that it is more difficult to relate the results back to the source code.

This detailed investigation of function behavior is called *Call Site Report* by PAT. It records for each call site of the instrumented functions how often it got called and time spent in this instantiation of the function. Execution times are

measured with a high-resolution timer. The results are available for each CPU used in the parallel program. The next version of PAT will allow to gather hardware counter statistics for instrumented functions as well.

3. Last, if a very detailed analysis of the program behavior is necessary, PAT also supports event tracing. The object instrumenter of PAT can also be used to insert calls to entry and exit trace routines around calls to user or library routines. Entry and exit trace routines can be provided in two ways:
 - The user can supply function-specific wrapper functions. The routines must be written in C, they must have the same number and same types of arguments as the routine they are tracing, and finally, the wrapper function name for a function *func* must be *func_trace_entry* for entry trace routines and *func_trace_exit* for exit trace routines.
 - If specific wrapper routines for the requested function are not available, PAT uses generic wrapper code which just records the entry and exit of the function in the event trace.

In addition, PAT provides extra tracing runtime system calls, which can be inserted in the source code and allow to switch tracing on and off, and to insert additional information into the trace (e.g., information unrelated to functions).

The tracing features of PAT were developed in a collaboration of Research Centre Jülich with CRAY Research. Our institute implemented all the necessary special wrapper functions for all message passing functions available on the T3E (MPI, PVM, and SHMEM and for both the C and Fortran interfaces) which record the message traffic in the event trace. In addition, we implemented a tool for converting the event traces contained in PIF files into VAMPIR trace format.

The major drawback of PAT's object instrumentation is the very low-level interface for specifying the functions to be instrumented. The user has to specify the function names as they appear in the object code, i.e., C++ functions or F90 functions which are local or contained in modules have to be specified in the mangled form (e.g., `__0FDfooPd` instead of the C++ function name `"int foo(double *)"`). Clearly, a more user friendly or automatic way for the instrumenter interface needs to be added to PAT.

In addition, the combination of three different instrumentation/analysis techniques into a single tool is very confusing for users. This confusion is further increased since the supported techniques overlap with the techniques applied in the other tools.

4.6 Summary

The previous sections pointed out that the CRAY T3E has a programming environment that includes the most advanced performance analysis tools. On the other hand, each of these tools comes with its own instrumentation, provides partially overlapping information, and has a totally different user interface. The programmer has to understand the advantages and disadvantages of all the tools to be able

	Apprentice	VAMPIR	PAT		
data collection	instrumentation via compiler	source instrumentation by preprocessor	sampling	object code inst.	
intrusion	high / corrected	high	low	high	
selective instr.	not required, optional	required	–	required	
selection interface	compiler switch	GUI or ASCII file	–	ASCII interface or file	
level of detail	summary information	event trace	statistics	summary information	event trace
information	total time and oper. counts for basic blocks and call sites	subroutine start/stop and send/recv events	statistical distr. of time (subroutines)	total time and pass counts for call sites	subroutine start/stop and send/recv events
strength	source-level, analysis of loops	many displays for message passing history and for statistics of arbitrary execution phases	low overhead profiling	total time for call sites, no recompilation	object code instr. for VAMPIR, no recompilation

Table 1. User interface and properties of performance analysis tools on CRAY T3E

to select and apply the right ones. Table 1 summarizes the main features of these three tools.

5 Summary

This article gave an overview of parallel programming models as well as programming tools. Parallel programming will always be a challenge for programmers. Higher-level programming models and appropriate programming tools only facilitate the process but do not make it a simple task.

While programming in MPI offers the greatest potential performance, shared memory programming with OpenMP is much more comfortable due to the global style of the resulting program. The sequential control flow among the parallel loops and regions matches much better with the sequential programming model all the programmers are trained for.

Although program tools were developed over years, the current situation seems not to be very satisfiable. Program debugging is done on a per thread basis, a technique that does not scale to larger numbers of processors. Performance anal-

ysis tools do also suffer scalability limitations and, in addition, those tools are complicated to use. The programmers have to be experts for performance analysis to understand potential performance problems, their proof conditions and their severity. In addition they have to be experts for powerful but also complex user interfaces.

Future research in that area has to try to automate performance analysis tools, such that frequently occurring performance problems can be identified automatically. It is the goal of the ESPRIT IV working group APART on *Automatic Performance Analysis: Resources and Tools* to investigate base technologies for future more intelligent tools ¹. A first result of the work is a collection of performance problems for parallel programs that have been formalized with the ASL, the *APART Specification Language* ⁶. This approach will lead to a formal representation of the knowledge applied in the manually executed performance analysis process and thus will make this knowledge accessible for automatic processing.

A second important trend that will effect parallel programming in the future is the move towards clustered shared memory systems. Within the GoSMP study executed for the Federal Ministry for Education and Research its influence on program development is investigated. Clearly, a hybrid programming approach will be applied on those systems for best performance, combining message passing between the individual SMP nodes and shared memory programming in a node. This programming model will lead to even more complex programs and program development tools have to be enhanced to be able to help the user in developing those codes.

References

1. APART: *ESPRIT Working Group on Automatic Performance Analysis Resources and Tools*, www.fz-juelich.de/apart, 1999.
2. D.P. Bertsekas, J.N. Tsitsiklis: *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, ISBN 0-13-648759-9, 1989.
3. Cray Research: *Cray MPP Fortran: Reference Manual*, Cray SR-2504, 1994.
4. D.E. Culler, J.P. Singh, A. Gupta: *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann Publishers, ISBN 1-55860-343-3, 1999.
5. L. Dagum, R. Menon: *OpenMP: An Industry-Standard API for Shared-memory Programming*, IEEE Computational Science & Engineering, 1998.
6. Th. Fahringer, M. Gerndt, G. Riley, J. Träff: *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, Research Centre Juelich FZJ-ZAM-9918, 1999.
7. I. Foster: *Designing and Building Parallel Programs*, Addison Wesley, ISBN 0-201-57594-9, 1994.
8. G.Fox: *Domain Decomposition in Distributed and Shared Memory Environments*, International Conference on Supercomputing June 8-12, 1987, Athens, Greece Lecture Note in Computer Science 297, edited by Constantine Polychronopoulos, 1987.
9. M. Gerndt, A. Krumme, S. Özmen: *Performance Analysis for SVM-Fortran*

- with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995.
10. MPI Forum: *Message Passing Interface*, www.mcs.anl.gov/mpi, 1999.
 11. W.E. Nagel, A. Arnold, M. Weber, H-C. Hoppe, K. Solchenbach: *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Vol. 12, No. 1, pp. 69-80, 1996.
 12. OpenMP Forum: *OpenMP Standard*, www.openmp.org, 1999.
 13. M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongarra: *MPI - The Complete Reference*, MIT Press, ISBN 0-262-69216-3, 1998.
 14. Etnus Inc.: *Totalview*, www.etnus.com/products/totalview/index.html, 1999.